Meri Rekiranta

Dataorientoituneen renderöijän toteuttaminen OpenGL:llä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

24.4.2012

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Meri Rekiranta Dataorientoituneen renderöijän toteuttaminen OpenGL:llä 25 sivua + 1 liite 24.4.2018
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tieto- ja viestintätekniikka
Suuntautumisvaihtoehto	Pelisovellukset
Ohjaaja(t)	Lehtori, Miikka Mäki-Uuro

Insinöörityön tavoitteena oli toteuttaa pohja tarpeiden mukaisesti laajennettavalle renderöijälle. Testausta varten toteutettiin 3D-mallin prosessointi, tuonti, vienti ja lataus peliin sekä piirto pelin Team Fortress 2 -tyyppisellä varjostuksella. Rajapinnalta vaadittiin kykyä OpenGL-grafiikkarajapintaa käyttäen piirtää reaaliaikaista kuvaa rajapinnan käyttäjän spesifioimalla tavalla.

Lopputuloksena saatiin aikaan reaaliaikagrafiikan piirtoa tarvitseville sovelluksille toimiva rajapinta, joka pystyy lataamaan yksinkertaisia varjostimia ja 3D-malleja. Tuotos toimii hyvänä pohjana tuleville peliprojekteille ja alkuna käyttötarkoituksia varten laajennettavalle pelimoottorille.

Avainsanat	OpenGL, dataorientoituneisuus, dataorientoitu ohjelmointi, varjostin, sheideri, renderöijä, rendaaja, grafiikkaohjelmointi

Author(s) Title Number of Pages Date	Meri Rekiranta Implementation of a Data-Oriented Renderer with OpenGL 25 pages + 1 appendix 24 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Games Software
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer

The goal of the bachelor's thesis was to implement a foundation for a renderer, that could be modified and extended to suit the needs of future projects. For testing purposes, exporting and importing of a 3D-modeled character in addition to the capability of drawing it with a Team Fortress 2 -like shading were implemented. It was required for the API to have the capability of using OpenGL to draw real-time graphics in the way that was specified by the user.

The end product was an API suited for drawing real-time graphics with the capability of loading and using simple shaders and 3D-models. The API works well as a foundation for future game projects and a good start for a game engine, that can be extended as needed.

Keywords	OpenGL, Data-oriented programming, Shader, Renderer

Sisällys

Lyhenteet ja käsitteet

Asiakas *Client.* Laite, joka lähettää palvelimelle pyyntöjä. Reaaliaikagrafiikan kontekstissa tämä tarkoittaa yleensä tietokoneen keskussuoritinta.

Beikkaus *Baking.* Graafisen efektin valmiiksi prosessointi 3D-malliin ennen ajoaikaa.

Fragmentti *Fragment.* Väridatan sisältävä piste kaksiulotteisessa ruutukoordinaatistossa. Fragmenttien avulla lasketaan ruudulla näytettävien pikselien väri.

Kaksoispuskurointi
Double buffering. Kahden kuvapuskurin käyttö grafiikan piirrossa.

Kuvapuskuri *Framebuffer.* Kaksiulotteisen pikselijonon sisältävä objekti, johon voidaan piirtää 3D-grafiikkaa.

Luu *Bone.* Tietorakenne, jota käytetään luustoanimaatiossa.

Luustoanimaatio
Skeletal animation. 3D-hahmojen ja -mallien animointitekniikka, jonka avulla verteksit voidaan liittää ”luihin” sen sijaan, että jokaista verteksiä tarvitsee erikseen animoida.

OpenGL Khronos Groupin spesifioima alemman tason rajapinta, joka abstrahoi grafiikkalaitteiston käyttöä.

Palvelin *Server.* Laite, joka suorittaa toimintoja asiakkaan pyynnöstä. Reaaliaikagrafiikan tapauksessa tämä tarkoittaa yleensä näytönohjainta.

Polygoniverkko
Polygon mesh. Toisiinsa yhdistetyistä vertekseistä koostuva verkko, jolla voidaan kuvata kolmiulotteisia objekteja tietokonegrafiikassa.

Pyyhkäisy *Render pass.* Yhden graafisen efektin piirto.

Renderöijä Rajapinta, joka abstrahoi grafiikan piirtämistä, grafiikkatiedon tuomista ja prosessointia ja ylemmällä tasolla grafiikkalaitteiston käyttöä.

SIMD	*Single Instruction, Multiple Data.* Prosessoriarkkitehtuurien luokka, joka pystyy suorittamaan saman operaation suurelle määrälle dataa samaan aikaan.
Sivunkääntö	*Page flipping*. Kaksoispuskuroinnin tekniikka, jossa kuvapuskureiden sisältö pystytään näyttämään ruudulla puskuria kopioimatta.
Skinnaus	*Skinning.* 3D-mallin liittäminen ”luihin”.
Spraitti	*Sprite.* Kaksiulotteinen bittikartta.
Tekseli	*Texel.* Pikseli 3D-mallin tekstuurissa.
Tekstuuri	*Texture.* Bittikartta, jota käytetään 3D-mallin pinnan värien spesifiointiin.
Varjostin	Näytönohjaimelle ladattava ja siellä suoritettava aliohjelma grafiikkadatan piirtoa varten.
Verteksi	*Vertex.* Piste avaruudessa.

1 Johdanto

Insinöörityön tavoitteena oli oman OpenGL-renderöijän perusominaisuuksien toteutus ja puurakenteisen modulaarisen pohjan luonti geneerisemmälle renderöijälle. Tämän kehityksessä hyödynnettiin dataorientoitunutta lähestymistapaa käyttäen spesifimpää tapausta esimerkkinä ja testauksen välineenä. Spesifisempänä kohteena tässä opinnäytetyöprojektissa toteutettiin sarjakuvamainen varjostus 3D-malleille. Lisäksi toteutuksen eri vaiheita ja efektejä analysoitiin pääpiirteittäin. Varjostuksen inspiraationa käytettiin Team Fortress 2 -pelin art stylea.

Tämä dokumentti käsittelee järjestyksessä ensin insinöörityön renderöintiin liittyviä peruskäsitteitä ja antaa lukijalle yleiskuvan grafiikkaohjelmoinnista ja sen haasteista. Kun lukija on tutustutettu aiheeseen, määritellään dataorientoitunut suunnittelu ja ohjelmointi sekä niiden perusperiaatteet. Tämän jälkeen periaatteet tunteva alan asiantuntija pystytään perehdyttämään itse insinöörityön aiheeseen eli renderöijän rakenteeseen. Rakennetta käsitellään suunnittelun kannalta ja myöhemmin itse toteutuksen näkökulmasta sekä esimerkkispesifisesti. Lopuksi lopputulosta käsitellään kohdassa Loppusanat.

2 Renderöinti

2.1 Renderöinti

Renderöinniksi kutsutaan prosessia, jossa matemaattisesta datasta ja kuvadatasta luodaan kolmiulotteinen kuva [1, s. 14]. Monissa nykyaikaisissa tietokoneissa on laitteisto – yleensä niin sanottu näytönohjain – joka pystyy piirtämään kolmiulotteista grafiikkaa reaaliajassa tietokoneen ruudulle. Tällaista kuvan piirtämistä kutsutaan laitteistokiihdytetyksi grafiikaksi.

Laitteistokiihdytettyä grafiikkaa piirtäviä sovelluksia ajavassa tietokoneessa näytönohjaimen ja suorittimen välille muodostuu palvelin-asiakas-suhde. Tämä tarkoittaa sitä, että näytönohjain eli ”palvelin” suorittaa suorittimen eli ”asiakkaan” spesifioimien ohjei-

den perusteella grafiikkadatan tai muun datan prosessointia ja piirtoa näytölle. [1, s. 18-19.]

Tyypillisessä tapauksessa kolmiulotteisen reaaliaikagrafiikan piirtämiseen käytetään valmiiksi luotuja ja prosessoituja tai proseduraalisesti generoituja 3D-malleja. Suosittu tekniikka 3D-mallien piirtämiseen on niin sanottu rasterointi.

Rasterointi on prosessi, millä vektorigrafiikka pystytään muuntamaan väreillä täytetyiksi muodoiksi kuvaruudulla. Koska merkittävä osa OpenGL-rajapinnan ominaisuuksista on rakennettu tekniikan ympärille, päätettiin sitä käyttää projektin renderöijän toteutuksessa. Tekstissä käsitellään renderöijää rasteroinnin näkökulmasta.

Renderöijä reaaliaikagrafiikan kontekstissa tarkoittaa rajapintaa, joka abstrahoi grafiikan piirtämistä, grafiikkatiedon tuomista ja prosessointia ja grafiikkalaitteiston käyttöä. Se toisin sanoen toimii kerroksena yksinkertaistamassa palvelimen ja asiakkaan välistä kommunikaatiota. Renderöijä sisältää usein myös työkaluja ja grafiikka-algoritmien toteutuksia, joita korkeamman tason sovelluskehittäjä voi käyttää apunaan.

Yleisesti ottaen renderöijä toteutetaan käyttäen apuna matalan tason grafiikkarajapintoja kuten Khronos Groupin spesifioimaa OpenGL:ää tai uudempaa Vulkania tai Microsoftin spesifioimaa Direct3D:tä. Näiden matalan tason grafiikkarajapintojen spesifikaatiot ovat vain listaus ominaisuuksista, jonka rajapinta pitää täyttää ja toteutukset ovat osa näytönohjainajuria ja erityisesti Direct3D:n tapauksessa yleisesti ottaen laitevalmistajien kirjoittamia. Spesifikaatioita ylläpitävät tahot lisensoivat laitevalmistajille rajapintansa nimen käyttöoikeuden ja pitävät huolta, että toteutukset täyttävät asetetut laatuvaatimukset. [1, s. 33-39.]

Vulkan sekä modernit versiot OpenGL:stä ja Direct3D:stä käyttävät ohjelmoitavaa piirtojärjestelmää. Tämä tarkoittaa sitä, että rajapinnat mahdollistavat näytönohjainmuistin joustavan hallinnan sekä näytönohjainaliohjelmien, eli varjostinohjelmien, lataamisen näytönohjaimelle ja niiden suorittamisen palvelimella.

Koska Direct3D-yhteensopivat näytönohjainajurit ovat käytännössä aina suljettua koodia ja rajoitettuja Microsoftin alustoille, päätettiin toteutus tehdä avoimempaa Khronos Groupin spesifioimaa rajapintaa käyttäen. Koska taaksepäin yhteensopivuus haluttiin säilyttää hyvänä, toteutukseen valittiin OpenGL Vulkanin sijaan.

Tämän opinnäytetyöprojektin spesifisessä tapauksessa renderöijä tarkoittaa rajapintaa, joka lataa 3D-mallin datan ensin välimuistiin, jossa sitä muutetaan ja prosessoidaan ja sen jälkeen ladataan näytönohjainmuistiin. Tämän jälkeen renderöijä lataa spesifioidut näytönohjaimelle mallin näyttöön piirrolle vaadittavat näytönohjainaliohjelmat eli varjostimet. Ajoaikana renderöijän tehtävä on toimia ylemmän tason koodin ja grafiikkalaitteiston välissä piirtämisessä ja grafiikkadatan muuntamisessa.

2.2 Grafiikkaresurssit

Ennen grafiikan piirtoa ohjelmoitavassa piirtojärjestelmässä tulee grafiikkaresurssit ladata palvelimelle. Renderöijän tehtäviin kuuluu grafiikkaresurssien ja niitten latauksen abstrahointi ylemmän tason ohjelmoijilta. Nämä resurssit voidaan jakaa eri kategorioihin luonteensa perusteella.

Attribuuttidata on luonteeltaan objektitaulukkomaista. Sen yksittäiset alkiot ovat yleensä lyhyitä, mutta pituudeltaan taulukot voivat hyvin olla gigatavujenkin kokoisia. Taulukoiden alkioita käytetään varjostimien syötteenä. Attribuuttidataa voidaan prosessoida muun muassa verteksidatana verteksivarjostinta käyttäen. Verteksidatassa taulukon alkiot kuvaavat verteksiä, eli pistettä kolmiulotteisessa avaruudessa, ja sen ominaisuuksia. Verteksejä käytetään monesti niin sanotun polygoniverkon luomiseen. Polygoniverkko on joukko toisiinsa yhdistettyjä verteksejä, jotka muodostavat kolmiulotteisesti simuloitavan objektin, eli niin sanotun 3D-mallin, pinnan.

Varjostimet ovat palvelimella suoritettavia ohjelmia, jotka on suunniteltu prosessoimaan suurta määrää erilaista syötettä saman operaation avulla. Käytännössä varjostinohjelma suoritetaan rinnakkaisesti niin, että palvelin prosessoi tuhansia syötteitä samanaikaisesti. Vaikka varjostimien käyttötavat ovat monipuolistuneet lähiaikoina, grafiikan piirtoon kuvaruudulle tarvitaan perinteisesti kaksi varjostinta: verteksivarjostin ja fragmenttivarjostin. Nämä yhdessä valinnaisen geometriavarjostimen kanssa muodostavat graafisen efektinohjelman, joka suoritetaan kutsumalla piirtokutsua. Piirtokutsua varten tulee spesifioida attribuuttidatan osa, joka halutaan piirtää.

Verteksivarjostin on ohjeistus palvelimelle verteksidatan prosessointiin. Verteksivarjostin ottaa syötteeksi näennäisesti yhden verteksin kerrallaan. Verteksivarjostin tulostaa

operaation jälkeen kolmiulotteisen paikkakoordinaatin ja skaalan sekä mahdolliset syötteet geometria- ja fragmenttivarjostimelle.

Geometriavarjostin verteksivarjostimen tapaan ottaa syötteeksi verteksidataa ja tulostaa kolmiulotteisia paikkakoordinaatteja. Verteksivarjostimesta poiketen geometriavarjostin prosessoi verteksidataa useampi verteksi kerrallaan. Se kuinka monta perättäistä verteksiä prosessoidaan kerralla, riippuu siitä, minkälaisia geometrisia primitiivejä palvelin on asetettu piirtämään. Kun paikkakoordinaatit on saatu määriteltyä, leikkautuu pois kaikki geometria, joka jää grafiikkarajapinnan määrittelemän heksaedrin ulkopuolelle, ja sen sisällä oleva geometria projisoidaan ortografisesti kaksiulotteiseksi kuvaksi. Geometria voidaan projisoida pisteiksi, rautalankamalliksi tai kiinteiksi primitiiveiksi, eli primitiiveiksi, jotka täyttävät värillään myös sisäänsä jättämän alueen kaksiulotteisessa kuvassa.

Fragmenttivarjostin on yksi varjostintyypeistä. Sen tehtävänä on prosessoida niin sanottuja fragmentteja. Fragmenttivarjostin käy läpi ohjelmoijan ennalta määrittelemin välimatkoin geometrian värittämän alueen kaksiulotteisessa kuvassa ja antaa jokaiselle prosessoimalleen kaksiulotteiselle koordinaatille, eli fragmentille, värin. Nämä värit piirretään sitten kaksiulotteiselle taulukolle, eli kuvapuskurille, jonka jokainen alkio, eli pikseli, saa koordinaattiaan vastaavan fragmentin/fragmenttien värin arvokseen. Palvelin voi myös asettaa jokaiselle fragmentille syvyysarvon, jonka se tallentaa niin sanotulle syvyyspuskurille. Syvyysarvo tarkoittaa kolmiulotteisen polygonin etäisyyttä kamerasta, eli sen z-koordinaattiarvoa. Mikäli syvyyspuskurointi on käytössä, piirtyy kuvapuskurille ainoastaan fragmentit, joiden syvyysarvo on pienempi kuin puskurissa olevan fragmentin syvyysarvo. Se kuinka monta fragmenttia pikseliä kohden on, ohjelmoija voi määrittää itse. Mikäli fragmentti ei ole verteksin päällä, ja geometria piirretään kiinteinä primitiiveinä tai rautalankamallina, verteksivarjostimen fragmenttivarjostimelle antamia syötteitä on syötetyyppiä kohden useita ja niitten arvot voidaan interpoloida lineaarisesti yhdeksi syötteeksi syötetyyppiä kohden. Yksi lineaarisesti interpoloitava syötetyyppi on niin sanottu tekstuurikoordinaatti.

Tekstuuri on taulukko väriarvoja tai väriarvon tapaisia arvoja, joita fragmenttivarjostin voi käyttää fragmentin värin määrittelyyn. Tyypillisessä tapauksessa fragmenttivarjostin saa verteksivarjostimelta syötteeksi tekstuurikoordinaatin, joka vastaa pyöristettynä kaksiulotteisen tekstuurin alkion indeksiä. Tämän alkion värinarvon avulla fragmenttivarjostin voi määrittää fragmentin värin. Tekstuuri voi myös toimia kuvapuskurina, eli

asiakas voi tarvittaessa asettaa palvelimen piirtämään grafiikkaa vakiokuvapuskurin sijaan määrittelemälleen tekstuurille. Tekstuuridata kategorisoidaan monesti uniformidatan yhtenä tyyppinä.

Uniformidata on varjostimille syötettävää dataa, joka pysyy yleisesti ottaen samana piirtokutsun ajan. Tämä tarkoittaa sitä, että mikäli varjostin saa syötteeksi uniformin, riippumatta siitä, mitä kohtaa verteksi-, fragmentti- tai attribuuttidatasta varjostin on prosessoimassa, syötetty uniformi pysyy samana. Uniformeja ovat muun muassa transformaatiomatriisit, joiden kanssa kertomalla verteksien ja sitä mukaa koko polygoniverkon sijaintia, orientaatiota ja skaalaa voidaan muuttaa kolmiulotteisessa avaruudessa.

2.3 Piirto

Kun data on ladattu palvelimelle, sitä käyttäen voidaan piirtää grafiikkaa ruudulle. Reaaliaikagrafiikassa tämä piirtoprosessi jaotellaan kuviin. Yksi kuva vastaa yhtä valmista staattista kaksiulotteista visualisaatiota kuvattavan maailman ajanhetkestä efekteineen. Usein kuvan piirtoon kuuluu useampia vaiheita.

Kuvan piirto aloitetaan yleensä kuva- ja syvyyspuskureiden tyhjentämisellä. Tyhjentäminen tapahtuu niin, että palvelin muuttaa taulukkojen jokaisen alkion arvon samanarvoiseksi, eli toisin sanoen täyttää niitten jokaisen pikselin samalla värillä. Tällä vältetään se, että aikaisemmin puskureille piirretty kuva vaikuttaisi piirrettävän kuvan lopputulokseen.

Tyhjille puskureille voidaan piirtää grafiikkaa pyyhkäisy kerrallaan. Pyyhkäisy tarkoittaa yhden efektin ja sillä piirrettävän grafiikan piirtoa. Monissa tapauksissa pyyhkäisyt piirretään erillisille kuvapuskureille, joita voidaan myöhemmissä vaiheissa käyttää tekstuureina.

Kun pyyhkäisyt on piirretty, voidaan ne koota ja piirtää yhteen kuvapuskuriin kompositointivaiheessa. Kompositoinnissa piirretään yleensä yksi kiinteä koko kuvapuskurin täyttävä neliö, johon fragmenttivarjostin koostaa lopullisen efektin.

Kompositoitua kuvaa voidaan viimeistellä jälkiprosessointiefekteillä. Jälkiprosessointiin kuuluvat esimerkiksi värinkorjaus. Värinkorjauksen tarkoituksena on muuttaa valmiin

kuvan jokaisen pikselin värin arvoa ennalta määritetyn hakutaulukon perusteella. Hakutaulukko sisältää jokaiselle värille sitä vastaavan toisen värin.

Reaaliaikagrafiikkasovelluksille on tyypillistä, että sen sisältämä liikkuva kuva ja animaatio piirretään samalla nopeudella, kuin mikä simuloitavan animaation nopeus on. Käytännössä tämä tapahtuu niin, että tietokoneen näyttäessä ruudulla yhtä kuvapuskurilla olevaa staattista kuvaa animaatiosta, tietokone piirtää samaan aikaan seuraavaa kuvaa. Mikäli tämä prosessi on tarpeeksi nopea, eli esimerkiksi kuvia pystytään piirtämään ja näyttämään ruudulla yli 30 sekunnissa, muodostaa tämä illuusion liikkuvasta kuvasta. Sen välttämiseksi, että kuvapuskurilla olevaa kuvaa näytettäisiin ruudulla samaan aikaan, kun siihen ollaan piirtämässä, moderneissa reaaliaikagrafiikkasovelluksissa käytetään kahta kuvapuskuria, joista yhtä puskuria näytetään ruudulla toiseen piirrettäessä. Tätä kutsutaan kaksoispuskuroinniksi. Nykyaikaisissa toteutuksissa käytetään niin sanottua sivunkääntötekniikkaa. Sen avulla molempia puskureita voidaan näyttää ruudulla, mikä tarkoittaa sitä, ettei piirrettyä kuvaa tarvitse kopioida kuvapuskurilta toiselle.

3 Dataorientoituneisuus

3.1 Dataorientoituneisuus

Työn toteutukseen valittiin erittäin dataorientoitunut lähestymistapa. Dataorientoituneisuus tarkoittaa, että ohjelmaa ajatellaan prosessina, joka käyttää, muuttaa ja tulostaa dataa. Tällaisen suunnittelun periaatteena on se, että ohjelmointiin liittyvän ongelman ymmärtämistä varten tulee ymmärtää se, mitä dataa ollaan prosessoimassa, mikä prosessoinnin suorituskustannus on ja minkälaisella laitteistoalustalla sitä ajetaan. Tämä tarkoittaa sitä, että algoritmien toteutus on riippuvaista käsiteltävän datan luonteesta sekä sitä prosessoivan prosessorin fyysisistä ominaisuuksista. [2.]

Objektiorientoituneessa ohjelmoinnissa on tyypillistä mallintaa data ja sen suhteet ja asettelu mahdollisimman lähelle sitä, mikä sen kuvaama oikean maailman konsepti on ja mikä sen suhde on maailmaan. Voidaan ottaa esimerkiksi tuoli. Oikeassa maailmassa ”tuolit” jakavat keskenään samanlaisia ominaisuuksia ja voidaan kategorisoida samaan kategoriaan. Tämä tarkoittaa sitä, että mikäli esimerkiksi pelisovellukseen halutaan lisätä erilaisia tuoleja, objektiorientoituneella lähestymistavalla on tyypillistä luoda

yliluokka ”Tuoli” ja siitä periytyvät aliluokat kuten ”Fysiikkatuoli”, ”StaattinenTuoli” ja ”RikkoutuvaTuoli”. Tässä lähestymistavassa on kuitenkin ongelma: nämä kaikki tuolit ovat sisältämänsä datan ja ominaisuuksiensa suhteen täysin erilaisia. Esimerkiksi ”Fysiikkatuoli” saattaa monessa kontekstissa olla datansa ja käyttötarkoituksensa suhteen lähempänä ”Fysiikkapöytää” samalla kun ”StaattinenTuoli” on lähempänä luokkaa ”Lattia”. Dataorientoitunut ohjelmointi ratkaisee tämän ongelman kategorisoimalla datan sen todellisten yhtäläisyyksien ja suhteiden perusteella. [2.]

3.2 Dataorientoituneisuus ja muistin optimointi

Performanssikriittisissä sovelluksissa on tyypillistä priorisoida sitä, että data on tarjolla prosessoitavaksi ja prosessoidaan mahdollisimman tehokkaasti niin, että mahdollisimman vähän ylimääräistä dataa joudutaan noutamaan välimuistiin. Tämän optimoinnin toteuttamisessa tulee ottaa huomioon se, mitä operaatioita datalle suoritetaan ja kuinka usein [2.]

Käytännössä tämä voi tarkoittaa esimerkiksi sitä, että pelissä on Pallo-nimisiä objekteja, jotka sisältävät paljon tietoa itsestään. Pelissä saatetaan tarvita esimerkiksi päivittää objektien paikkakoordinaatteja tietyn ajan välein. Perinteisellä objektiorientoituneella lähestymistavalla tämä ratkaistaisiin Esimekkikoodi 1:n mukaisesti.

```
typedef struct
{
    Paikka paikka;
    float koko;
    Vari vari;
    MyEnum tyyppi;
} Pallo;

void PaivitaPallo(Pallo *pallo, double deltaAika)
{
   ...
}

void PaivitaPallot(Pallo *pallot, size_t numPallot, double
deltaAika)
{
   usigned int i;
   for(i = 0; i < numPallot; ++i)
   {
      PaivitaPallo(pallot + i, deltaAika);
   }
}
```

Esimerkkikoodi 1. Objektiorientoitunut lähestymistapa Pallojen päivittämiseen.

Tässä kuitenkin muodostuu ongelmaksi se, että pallon kaikki data on muistissa peräkkäin. Tämä tarkoittaa, että PaivitaPallo aliohjelma lataa välimuistiin ei pelkästään paikka-muuttujan vaan mahdollisesti kaiken muunkin datan, mitä pallot[i] sisältää. Tähän ongelmaan on kuitenkin ratkaisu dataorientoituneessa ohjelmoinnissa, jota illustroi Esimerkkikoodi 2:

```
typedef struct
{
    Paikka *paikat;
    float *koot;
    Vari *varit;
    MyEnum *tyypit;
    size_t numPallot;
} Pallot;

void PaivitaPallonPaikka(Paikka *paikka, double deltaAika)
{
   ...
}

void PaivitaPallot(Pallot *pallot, size_t numPallot, double
deltaAika)
{
   usigned int i;
   size_t numPallot;
   Paikka *paikat = pallot->paikat;
   numPallot = pallot->numPallot;
   for(i = 0; i < numPallot; ++i)
   {
      PaivitaPallo(paikat + i, deltaAika);
   }
}
```

Esimerkkikoodi 2. Dataorientoitunut ratkaisu paikkadatan päivittämiseen.

Dataorientoituneessa ratkaisussa jokaisen pallo-objektien jäsenet on asetettu muistissa niin, että peräkkäin prosessoitavat datat on asetettu muistissa peräkkäin vähentäen välimuistihuteja. Tällainen datan asettelu helpottaa myös monisäikeistämistä ja datan siirtelemistä järjestelmämuistin ja näytönohjainmuistin välillä. Modernit näytönohjaimet toimivat ns. SIMD (eng. Single Instruction, Multiple Data) -piirin tavoin, eli niin, että se pystyy samaan aikaan toteuttamaan saman operaation rinnakkaisesti suurelle määrälle dataa. Tämä tarkoittaa sitä, että monisäikeistämisen kannalta yksinkertaiset datan asettelut helpottavat datan käyttöä näytönohjaimessa.

4 Renderöijän arkkitehtuuri

4.1 Suunnittelu

Renderöijän arkkitehtuuria lähdettiin kirjoittamaan aluksi point-and-click-teemaisen pelin toteuttamista varten. Alkuperäinen idea projektissa oli luoda suppea ja erikoistunut renderöijä kaksiulotteisille bittikartoille eli spraiteille. Sen oli tarkoitus sisältää niin sanottu sprite batcher, eli kyky lajitella ja järjestellä piirrettävät spraitit graafisten efektiensä ja piirtojärjestyksensä perusteella. Tämän ansiosta piirtoaikana laitteisto pystyy piirtämään halutun kokonaisuuden mahdollisimman vähillä piirtokutsuilla. Tällainen prosessointi palvelinkutsujen – ja erityisesti piirtokutsujen – suhteen on tärkeää renderöinnin optimoinnissa, sillä kommunikaatio asiakkaan ja palvelimen välillä on hidasta suhteessa moneen muuhun prosessointiin ja operaatioihin, joita laitteisto voi suorittaa peli- ja reaaliaikagrafiikkasovelluksissa.

Spraitit ovat OpenGL:n kontekstissa kaksi- tai joskus kolmiulotteisia nelikulmioita, joiden tekstuurit ovat usein osittain läpinäkyviä. Niiden piirto tapahtuu monin tavoin samalla tavalla kuin 3D-mallien piirto. Spraitti koostuu neljästä verteksistä, joitten kolmas ulottuvuuskoordinaatti on usein nolla. Se voidaan projisoida kuvapuskurille ortografisesti suoraan, ja kaikki sen tekstuurin pikselit, joiden alfa-arvo on lähellä nollaa, voidaan jättää piirtämättä. Tässä piirrossa on kuitenkin optimoinnin kannalta haasteita. Merkittävimpänä haasteena on se, että spraitteja voidaan tietynlaisissa peleissä piirtää jopa tuhansia samaan aikaan. Jos jokainen spraitti piirrettäisiin omana 3D-mallina omalla varjostimella, tulisi jokaista spraittia kohden yksi piirtokutsu, ja jopa yksinkertainen peli saattaisi vähemmän tehokkailla alustoilla aiheuttaa pullonkaulan optimoinnin kannalta. Sprite batcher ratkaisee tämän ongelman.

Sprite batcher kirjaa ennen piirtoa kuvan kaikki käyttäjän spesifioimat spraitit efekteineen, tekstuureineen, koordinaatteineen, prioriteetteineen ja transformaatioineen. Ennen piirtoa batcher katsoo, mitkä spraiteista voidaan piirtää samaan aikaan. Tällaisia ovat spraitit, joilla on samanlainen prioriteetti ja efekti. Tämän jälkeen nämä samaan aikaan piirrettävät spraitit asetellaan piirtolistassa peräkkäin, jonka jälkeen nämä samalla prioriteetilla ja efektillä piirrettävät mallit voidaan asetella peräkkäin sen perusteella, onko niillä sama tekstuuri. Nykyaikaisissa grafiikkalaitteistoissa on useita tekstuuripaikkoja, joihin tekstuuriuniformi voidaan yhdistää, joten vaikkei spraiteilla olisikaan samaa tekstuuria, voidaan ne piirtää silti samalla piirrolla. Yleensä sprite batcher sisältää pit-

kän listan neliön muotoisia koordinaateiltaan identtisiä kolmiulotteisia muotoja, joilla on erillinen attribuutti, joka kertoo, monennettako spraittia nelikulmion verteksi vastaa. Kun batcher on valmis spraittien lajittelussa, se lataa uuden version tästä attribuuttijonosta palvelimelle. Tämän attribuuttiarvon perusteella varjostimet tietävät, mitkä uniformit kuuluvat prosessoimalleen spraitille.

Alkuperäinen versio renderöijästä tehtiin suoraan monisäikeistetysti niin, että yksi säie suoritti muun pelilogiikan sekä kuvien piirtojärjestyksen määrittelyn, kun toinen säie suoritti itse piirron ja palvelinkutsut. Tuotos oli kuitenkin liian joustamaton, joten siitä jouduttiin luopumaan.

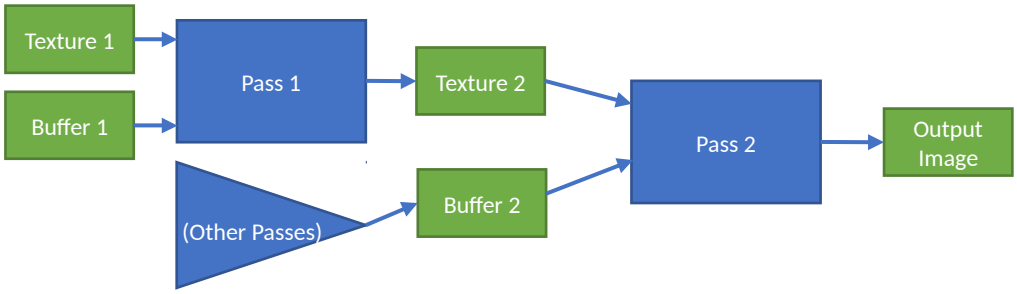
Seuraavat iteraatiot kehittivät monisäikeistystä eteenpäin niin, että kokonaisuutta geneeristettiin. Tämä tapahtui useassa iteraatiossa ja lopputuloksena syntyi järjestelmä, jossa prosessointisäikeet määrittelivät komentojonoon grafiikkakutsut ja niiden syötteet samalla, kun yksi säie kävi jonoa läpi ja kutsui jonon perusteella jo määriteltyjä kutsuja. Vasta jälkeenpäin huomattiin, että toteutus vastaa hyvin pitkälti asiakaspuolella toteutettua yksinkertaista versiota Vulkan-spesifikaatiossa valmiiksi määritellystä komentojonojärjestelmästä [3, s. 6-7].

Toteutuksen iterointi vei merkittävästi aikaa, joten renderöijää ei loppujen lopuksi käytetty peliprojektin toteuttamiseen ja lopullinen peli tehtiin Unity-pelimoottorilla. Renderöijän iterointi kuitenkin jatkui. Sprite batcherin toteutusta ei jatkettu insinöörityöprojektia varten ja päätettiin toteuttaa vasta projektin valmistuttua.

4.2 Lopullinen arkkitehtuuri

Pitkän suunnittelun ja testauksen jälkeen päätettiin lähteä toteuttamaan renderöijän arkkitehtuuria puurakenteena, jonka solmuina oli varjostimista inspiraation saaneet aliohjelmat, jotka ottavat syötteekseen suuria määriä dataa ja tulostavat prosessoidun datan ulos. Ideaalitapauksessa yhdellä solmulla on vain yksi ylisolmu, mutta jokaisella solmulla voi olla useampia alisolmuja. Aliohjelmien suoritus tapahtuu niin, että jokainen solmu ottaa alisolmujensa palauttaman datan syötteekseen, ja palauttaa sitten itse omalle ylisolmullensa dataa prosessoitavaksi. Tässä mallissa operaatioiden ja renderöintipyyhkäisyjen synkronisointi on yksinkertaista: alisolmujen aliohjelmat täytyvät olla valmiita ennen kuin ylisolmun aliohjelma voidaan suorittaa, joten järjestys tiedetään val-

miiksi. Nicholas Guillermot [4] esitelmässään kuvaa vastaavanlaista rakennetta Kuvan 1 mukaisesti.



Kuva 1. Puurakenteisen renderöijän arkkitehtuuri [4].

Paljon prosessointia vaativissa sovelluksissa renderöijän toiminta suositellaan perinteisesti monisäikeistettäväksi niin, että kaikki renderöijärajapinnan kutsut tapahtuvat samassa säikeessä, kun taas muu prosessointi tapahtuu erillisissä muissa säikeissä. Käytännössä tämä tarkoittaa sitä, että muut säikeet valmistelevat datan ja komennot komentojonoon, jonka pääsäie käy läpi siinä vaiheessa, kun muut säikeet ovat lopettaneet siihen kirjoittamisen. Uudemmissa grafiikkarajapinnoissa kuten Vulkanissa ja Direct3D 12:sta tämä komentojono on kuitenkin toteutettu rajapinnan itsensä puolesta, ja toisin kuin aikaisemmin, on tarkoitettu, että jokainen säie pystyy kutsumaan rajapintakutsuja [3, s. 1, 6-8, 11].

Vaikka renderöijäarkkitehtuurin kannalta on tärkeää ottaa huomioon ongelmat monisäikeistämisessä, ottaen huomioon projektin suppeuden, ei moderneissa laitteissa toteutuksen jättäminen yksisäikeiseksi merkittävästi vaikuta lopputulokseen. Tämän vuoksi vaikka aiemmissa iteraatiossa monisäikeistäminen oltiin toteutettu, jätettiin ominaisuus pois. Tarkoituksena oli kuitenkin jättää hyvä pohja sen mahdolliseen toteuttamiseen myöhemmin.

Lähestymistavassa on kuitenkin haasteita, joista monia vaikeuttaa vielä ajatus siitä, että projektin pitäisi olla hyvä pohja myöhemmin monisäikeistettävälle koodille. Sovelluksessa voisi hyvin olla resurssi, jota luetaan 200 kertaa, mutta johon kirjoitetaan vain kerran. Olisi resurssien kannalta hinnakasta katsoa joka kerta, onko kyseinen resurssi päivitetty ennen kuin sitä luetaan saati että tekisi erillistä versiota samasta resurssista. [5.]

5 Esimerkki

5.1 Toteutettu esimerkki

Idea esimerkin toteutuksesta lähti erään yrityksen työpaikkahakemukseen liittyvästä tehtävästä. Tehtävässä oli tarkoituksena toteuttaa Team Fortress 2:sta tuttu varjostustyyli toteutettuna Unity-pelimoottorilla. Kuitenkin, koska valmista pohjaa renderöijälle oltiin jo toteutettu, päätettiin toteutus tehdä sillä Unityn sijasta. Tuotos saatiin viikossa valmiiksi ja sen sekä erityisesti sen alla olevan renderöijätoteutuksen iterointi jäi elämään.

Esimerkin tarkoituksena oli olla taidetyyliltään Team Fortress 2:sta inspiraatiota ottava sarjakuvamainen varjostus, mutta voimakkaammalla spekulaarivalaistuksella. Niin sanottuina ice-to-have-ominaisuuksina sen lisäksi oli mallien ääriviivojen paksuntaminen ja dynaaminen varjostus. Lisäksi mikäli aikaa olisi jäänyt, parantelua oltaisiin voitu tehdä esimerkiksi lisäämällä erilaisia varjostimen permutaatiovaihtoehtoja, missä olisi esimerkiksi eri versio riippuen siitä, onko malli skinnattu vai ei.

Skinnaus 3D-grafiikassa tarkoittaa 3D-hahmon polygoniverkkojen verteksien kytkemistä ”luihin”, eli paikka- ja orientaatiodatan sisältäviin objekteihin. Monimutkaisemmissa tapauksissa, kuten ihmismalleissa, luut asetetaan yleensä hierarkiaan, jossa ylemmän hierarkiatason luiden orientaatio ja sijainti vaikuttavat alempien tasojen luiden orientaatioihin ja sijainteihin. Verteksin kytkeytyminen luihin tarkoittaa sitä, että luun sijainti ja orientaatio vaikuttavat verteksidatassa määrätyn kertoimen eli niin sanotun painotuksen verran verteksin sijaintiin. Tämä tarkoittaa, että mikäli 3D-hahmoa tarvitsee animoida, jokaisen verteksin sijaintidataa ei tarvitse erikseen määritellä vaan voidaan vain määritellä luut ja niitten suhteet ja vaikutukset mallin vertekseihin ja animoida ne.

Verteksien sijaintien muuttaminen voidaan tehdä yksinkertaisella algoritmilla verteksija luudataa käyttäen. Tämä algoritmi voidaan suorittaa laitteen asiakaspuolella, eli niin sanotusti ohjelmistoskinnata, mutta nykyään 3D-mallin verteksien sijainnin muuttaminen luiden mukaisesti tehdään verteksivarjostimessa, joten se on hyvä ottaa huomioon mallin efektejä toteuttaessa. Tätä kutsutaan laitteistoskinnaukseksi. [6, s. 49.]

Suunnitelmana oli kirjoittaa renderöijä ensin käyttökuntoon lähteä toteuttamaan esimerkkiä efekti efektiltä päivittäen ja testaten ohjelman alemman tason ominaisuuksia samaan aikaan.

5.2 Esimerkin valaistus ja efektit

Team Fortress 2:n mattapintaisen sarjakuvamaisen valaistuksen perusominaisuuksien toteuttaminen onnistuu 3D-reaaliaikagrafiikalle hyvin tyypillisillä tekniikoilla. Klassisempi tapa 3D-reaaliaikagrafiikassa reaalimaailman valon käyttäytymisen simulointiin sen heijastuessa objektista on jakaa heijastuva valo kolmeen kategoriaan: ambientti valaistus sekä spekulaari- ja diffuusiovalo. Tällaisen valaistuksen piirto tehdään yleensä varjostimessa, ja joissain tapauksissa erillisissä pyyhkäisyissä. Toteutustamme varten kuitenkin nämä kaikki voitiin prosessoida samassa varjostimessa yhdellä pyyhkäisyllä.

Ambientti valaistus simuloi tilannetta, missä objektin jokainen pinta valaistuisi samalla intensiteetillä riippumatta sen suunnasta ja yleensä myös etäisyydestä objektiin. Tämä ei siis ole sama kuin ambientin valon käsite reaalimaailmassa, missä vaikka valoa tulisi objektia kohden joka suunnasta, objekti saattaa muodoillaan estää tietystä suunnista tulevien valoaaltojen osumista tietyille pinnoille. Tämä aiheuttaa efektin, jota kutsutaan ambientiksi okkluusioksi, jossa malli luo itseensä pehmeitä varjoja peittämällä osan pinnoistaan. Staattisissa malleissa ambientti okkluusio voidaan kuitenkin beikata, eli laskea informaatio valmiiksi tekstuuriin, joten tässä esimerkkitapausta varten siitä ei tarvinnut huolehtia.

Ambientin valaistuksen toteuttaminen varjostimessa on teoriassa yksinkertaista. Fragmenttivarjostimelle annetaan vakioarvo, jolla kaikki tekselit, eli 3D-mallin tekstuurin pikselit, kerrotaan. Kuitenkin pelisovelluksille tarkoitetutuissa 3D-malleissa tekstuuri on sRGB-väriavaruudessa, eli sen värit muutettiin RGB:ksi eli väriarvoksi, jonka arvot vastaavat punaisen, vihreän ja sinisen todellista intensiteettiä havaitsijan näkökulmasta, vasta väriarvojen siirtyessä näytönohjaimelta näytölle. Suurimmalle osalle visuaalisista efekteistä on algoritmin kannalta yksinkertaisempaa työskennellä RGB-avaruudessa, joten ennen operaation suorittamista varjostimessa, täytyy muistaa muuntaa mallin väritekstuuri sRGB-avaruudesta RGB-avaruuteen ja fragmenttivarjostimen palauttamat väriarvot jälleen sRGB:ksi ennen näytölle piirtoa. Tähän muunnokseen löytyy suurim-

masta osasta moderneja näytönohjaimia valmis muuntaja, joka kuuluu OpenGL-rajapinnan ydinominaisuuksiin, joten muunnosta ei tarvitse itse kirjoittaa varjostimeen.

Diffuusiovalo simuloi tilannetta, missä täysin mattapintaista objektia valaistaisiin jostakin tietystä suunnasta. Tämä aiheuttaisi sen, että objektin ne pinnat, joiden tangentit osoittaisivat suoraan valonlähteeseen, valaistuisivat eniten, kun taas kaikki pinnat, joiden tangentit olisivat suuremmassa kuin 90 asteen kulmassa valonlähteen suhteen, jäisivät täysin valaistumatta. Tässä approksimaatiossa valkoisella valolla valaistuna objektista siroutuvan valon havaitun värin saturaatio on sama kuin pinnan oikea värin saturaatio riippumatta valonlähteen intensiteetistä niin kauan kuin heijastuvan valon intensiteetti ei mene havainnoitsijan havainnointikyvyn yli. Oikea värin saturaatio tarkoittaa tässä kontekstissa käänteistä suhdetta siihen, minkä verran punaista, vihreää ja sinistä valoa pinta absorboi. Tässä tapauksessa tämä tarkoittaa 3D-mallin väritekstuurin tekseliväriä muunnettuna RGB-väriavaruuteen. Havainnointikyky tarkoittaa tässä yhteydessä sitä, että ihmissilmä pystyy havaitsemaan samaan aikaa vain rajatun intensiteettivälin sisällä olevia valon intensiteettejä. Nykyaikaisemmissa realistisuutta hakevissa reaaliaikagrafiikkasovelluksissa kuitenkin pyritään pitämään tämä valon intensiteetti rajoittamattomana ja pyritään säätämään dynaamisesti intensiteettiväliä, joka on sillä hetkellä optimaalisin suurimman osan sillä hetkellä havainnoitavien asioiden näkemiseksi oikeissa väreissään. Tätä tapaa tehdä asioita kutsutaan nimellä High Dynamic Range eli HDR. Kuitenkaan tähän tarkoitukseen sen toteuttamista ei tarvita.

Spekulaarivalo tarkoittaa valoa, joka heijastuu pinnoista diffuusiovaloa pienemmällä sironnalla muodostaen valonlähteen värisen heijastuman pintoihin, jotka ovat lähellä valonlähteen heijastumapintaa suhteessa katsojaan.

Toisin sanoen tämä tarkoittaa sellaisia pintoja missä skalaarikolmiotulo

$$\frac{a \times b}{|a \times b|} \cdot c \approx 0 \tag{1}$$

ja

$$c \cdot a \approx -c \cdot -b \tag{2}$$

, jossa c on pinnan normaali, a on normalisoitu valonlähteen sijainti suhteessa pintaan ja b on normalisoitu katsojan sijainti suhteessa pintaan.

Tämä efekti saa aikaan sen, että 3D-mallin pinta näyttää kiiltävältä.

6 Toteutus

6.1 Ohjelmointikielen valinta

Tässä luvussa käytetään sanaa ”turvallisuus”. Suomessa sana vastaa sekä englannin sanaa ”safety” että ”security”. ”Safety” tarkoittaa tietotekniikkakontekstissa toimivuutta ja sitä, että jonkin asian toimimattomuus tai erikoistilanteet eivät saa aikaan sitä, että jokin rikkoutuisi tai tärkeää dataa menetettäisiin. ”Security” puolestaan tarkoittaa sitä, ettei luvatonta sisäänpääsyä tapahdu tai ettei asioita tai dataa joudu vääriin käsiin.

Renderöijä on reaaliaikaisessa sovelluksessa hyvinkin matalan tason komponentti ja sen takia myös yksiä performanssikriittisimmistä osista, joten ohjelmointikielen valinnassa merkittävä tekijä on sen tehokkuus. Toisaalta useimmat parempiluokkaiset reaaliaikagrafiikan sovellukset eivät usein ole niin turvallisuuskriittisiä kuin monien muitten alojen sovellukset (esim. tietoturva ja back-end -ohjelmointi). Tämän vuoksi voidaan valita kieli, jossa ohjelmoijalla on enemmän vastuuta ohjelmansa turvallisuudesta, mutta myös sitä kautta pystyy itse hallitsemaan monissa muissa sovelluksissa automaattisesti tapahtuvia asioita. Näistä tärkeimpinä mainittakoon roskienkeruu ja muu muistinhallinta. Hallitsemalla näitä asioita itse saadaan vähennettyä automaattisten ohjelmointikielispesifisten geneerisempien toteutusten optimointipullonkauloja, toteuttamalla itse ohjelmaspesifisemmän toteutuksen. Useimmiten ”securityä” enemmän ongelmia erityisesti pelisovelluksissa ja spesifisemmin renderöijässä aiheuttaa ”safety”. Renderöijä itsessään toimii hyvin vähän sensitiivisen tiedon parissa, kun taas sen luonne alhaisen tason komponenttina saa aikaan potentiaalista epävakautta.

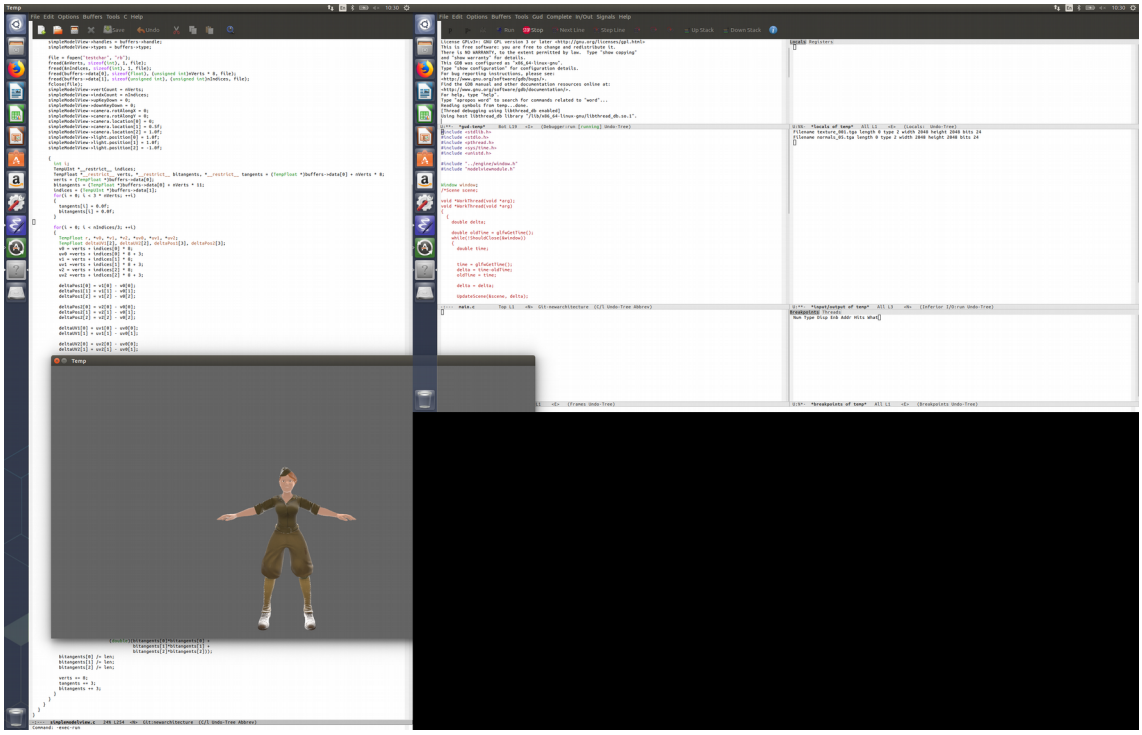
Pelisovelluksissa on monesti myös tärkeää kielen yhteensopivuus yleisesti alalla käytettyjen kirjastojen kanssa, joten alalla yleisesti käytettävät kielet tai yhteensopivuus niiden kanssa on merkittävä tekijä.

Nämä kriteerit huomioon ottaen ehdotettiin kolmea kieltä: C, C++ ja Rust. Koska kyseessä oli poikkeuksellisen dataorientoitunut ohjelma, ei objektiorientoituneita ominaisuuksia tarvittu niin vahvasti, joten se ei ollut merkittävä tekijä, joten päädyttiin C:hen sen yksinkertaisuuden takia. Vaikka C++:ssä ja Rustissa onkin paljon eri ohjelmointia

helpottavia ominaisuuksia, joilla ei ole merkittävää overheadia, projektin ja oppimisen kannalta C antaa paremman kokonaiskuvan siitä, mitä ohjelma tekee matalalla tasolla.

6.2 Työkalut

Toteutus tapahtui kokonaan GNU/Linux-alustalla, eikä tavoitteisiin kuulunut toteutukset muilla alustoilla. Koodin kirjoittaminen tapahtui GNU Emacsilla käyttäen evil-modea ergonomisuuden parantamiseksi ja editoinnin helpottamiseksi. GNU Emacsissa on laaja tuki ohjelmointityökaluille ja sen kautta käytetään myös kääntäjää ja gdb-debuggaustyökalua. Kuvassa 2 on kuvakaappaus työympäristöstä.



Kuva 2. Projektin työympäristö. Vasemmalla pystynäyttö, jossa takana GNU Emacs -ikkuna ja ohjelmakoodi ja sen edessä projekti itse. Oikealla vaakanäyttö ja toinen Emacs-ikkuna jossa gdb-many-windows -debuggaustyökalu.

C tuo projektiin muutamia haasteita, joita varten lisätyökalut ovat tarpeen. C ei ole turvallinen kieli ja sisältää itsessään hyvin vähän ominaisuuksia, jotka estäisivät ohjelmoijaa tekemästä hyvin kriittisiäkin virheitä koodissaan. Tämä tarkoittaa käytännössä, että huonosti kirjoitettu ilman vankilointia (eng. ”jailed”) oleva C-sovellus voi hyvin poistaa koko kotihakemiston tiedostot tai kaataa koko käyttöjärjestelmän. Kompensoimaan C-kielen turvallisuuden puutetta gdb:n perusominaisuuksien lisäksi käytetään Valgrindia muistinvuotojen löytämiseen ja muistin debuggaukseen sekä nykyään gcc:hen lisättyjä

työkaluja kuten AddressSanitizeria ja UndefinedBehaviorSanitizeria, jotta helposti huomaamatta jäävät ja harvoin toistuvat kaatumiset ja ongelmat saadaan korjattua.

Valgrind on muistin debuggaukseen, muistivuotojen löytämiseen ja ohjelman prosessoripuolella tapahtuvien operaatioiden profilointiin käytettävä työkalu GNU/Linuxille [7]. Projektissa sitä oli tarkoitus käyttää kaikkiin kolmeen tarkoitukseensa.

Ohjelman näytönohjaimessa suoritettavan toiminnallisuuden debuggaus ja profilointi täytyi kuitenkin tehdä erillisellä työkalulla. Tähän on useita vaihtoehtoja tarjolla. Aiemman kokemuksen perusteella parhaimmat ovat kuitenkin suljetun koodin ohjelmistoja kuten Nvidian grafiikkadebuggeri. Linuxilla Nvidian grafiikkadebuggeri vaatii varsin omituisia ratkaisuja kuten lokaalin ssh-serverin pystyttämistä, jotta debuggeri pääsee ohjelmaan käsiksi [8]. Tämän vuoksi sitä ei voi suositella ohjelmoijille, joille ssh ei ole aikaisemmin tuttu. Näitten syitten takia grafiikkadebuggaus suoritettiin muilla tavoin.

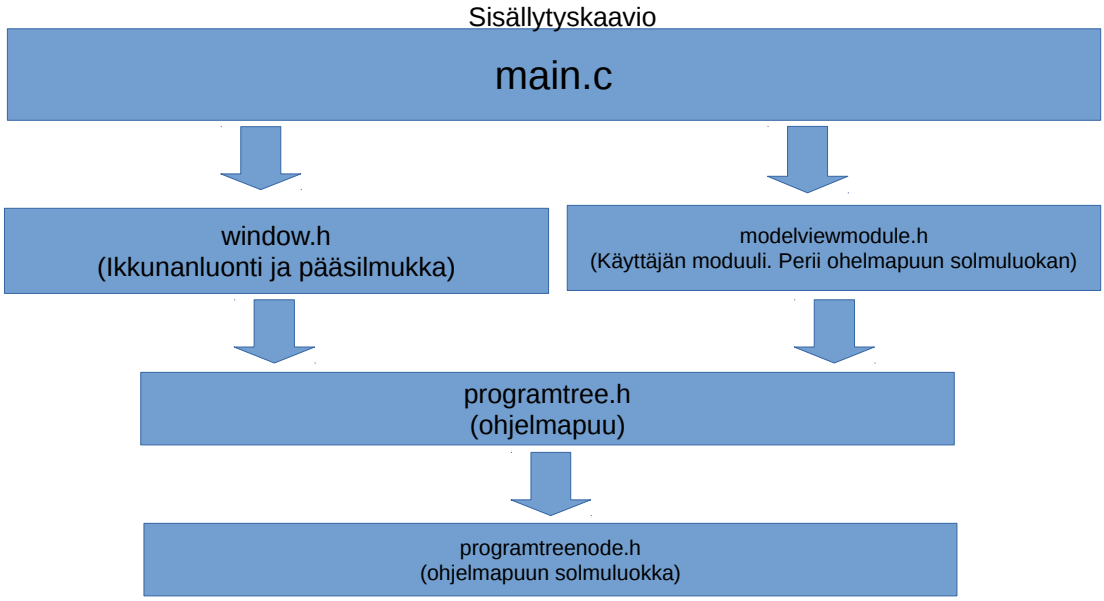
AddressSanitizer-työkalun avulla ohjelmoija pystyy testaamaan ja löytämään koodistaan kohtia, joissa ohjelma saattaisi hakea dataa allokoimattomista tai jo vapautetuista muistiosoitteista. Tämä tapahtuu analysoimalla eri kohtia koodista ja määrittelemällä mahdolliset syötteet, jota aliohjelmat voisivat saada. AddressSanitizerin toiminnallisuudessa on jonkin verran päällekkäisyyttä Valgrindin kanssa, mutta molemmilla työkaluilla pystyy havaitsemaan joitain sellaisia ongelmia, mitä toisella ei pysty.

6.3 Ikkunanluonti ja syötteet

Nykyajan tyypillisille laitteistoille ja käyttöjärjestelmille reaaliaikagrafiikkasovellusta kirjoittaessa aloitetaan monesti ikkunanluonnista. Käyttöjärjestelmillä on oma tapansa ja rajapintansa sovellukselle ikkunanluontia varten, ja näitä rajapintoja apuna käyttäen pystytään luomaan laitteistokiihdytetty OpenGL-konteksti ja kuvapuskuri ja spesifioimaan, minkälaisen ikkunan käyttäjä tarvitsee. Koska GNU/Linuxissa, Mac OS X:ssä ja Windowsissa on omat rajapintansa tähän, päätettiin käyttää valmiiksi tehtyä rajapintaa nimeltä GLFW ikkunanluonnin abstrahointiin. GLFW:ssä oli sen lisäksi myös ominaisuuksia käyttäjäsyötteen käsittelyyn, jota hyödynnettiin testauksessa.

6.4 Renderöijäarkkitehtuurin ytimen toteuttaminen

Toteutus alkoi renderöijän perusrakenteen luonnista. Yksinkertaisuudessaan aliohjelmapuu oli vain yksi lyhyehkö funktio, joka kävi läpi jokaisen solmun järjestyksessä aloittaen lehtisolmusta ja päättyen juureen. Jokaiselle solmulla oli mahdollisuus luoda itselleen lokaali muisti ja tarvitsemansa määrä alisolmuja. Solmu hoiti myös aliohjelmansa tulostaman datan allokoinnin, ellei se ollut suoraan alisolmuilta tullutta dataa. Kuva 3 illustroi arkkitehtuurin kokonaisrakennetta sisällytyskaaviolla.



Kuva 3. Ohjelman sisällytyskaavio.

Käytännössä toteutus tapahtui niin, että tehtiin ProgramTree-niminen C-luokka, joka sisälsi viittauksen objektiin tyyppiä ProgramTreeNode sekä tietoa puun hallinnointiin. ProgramTreellä oli funktio StartProgramTree() ProgramTreen puun aloittamiseen ja allokointiin. Ennen funktion StartProgramTree() kutsumista käyttäjällä oli mahdollisuus lisätä ProgramTreeNode-luokasta perittyjä objekteja ProgramTreehin. StartProgramTree():tä kutsuttaessa funktio kutsuu juuresta aloittaen ProgramTreeNodeen aloitus ja allokointifunktion LoadProgram(), joka puolestaan voi luoda alisolmuja ja kutsua niitten LoadProgram()-funktiota. Puun latauduttua ProgramTree kutsui funktiota UpdateProgramTree() jokaisella ruudulla, joka puolestaan kutsui ProgramTreeNoden päivitysfunktiota, aloittaen lehtisolmuista ensin. UpdateProgramTree():n logiikkaa illustroi Esimerkkikoodi 3.

```
void UpdateProgramTree(ProgramTree *programTree, float deltaTime)
{
  ProgramTreeNode **currentBranch;
  int *branchesUpdated;
  int currentBranchLevel = 0;
  currentBranch = programTree->currentBranch;
  branchesUpdated = programTree->branchesUpdated;
  for(currentBranchLevel = 0; currentBranchLevel < 32; +
+currentBranchLevel)
    branchesUpdated[currentBranchLevel] = 0;
  currentBranchLevel = 0;
  while(currentBranchLevel >= 0)
  {
    if(branchesUpdated[currentBranchLevel] ==
currentBranch[currentBranchLevel]->nChildren)
    {
      (*(currentBranch[currentBranchLevel]->program))
(currentBranch[currentBranchLevel]->locals,
currentBranch[currentBranchLevel]->out);
      branchesUpdated[currentBranchLevel--] = 0;
    }
    else
    {
      currentBranch[currentBranchLevel + 1] =
currentBranch[currentBranchLevel]->

children[ branchesUpdated[currentBranchLevel] ];
      ++branchesUpdated[currentBranchLevel];
      ++currentBranchLevel;
    }
  }
}
```

Esimerkkikoodi 3. UpdateProgramTree():n logiikka testivaiheessa.

Ensimmäinen toimiva versio oli tarkoitus luoda yhdellä solmulla toimivaksi. Tämä solmu ensin latasi piirrettävän datan, varjostimet ja muun relevantin piirtoa varten yhdessä aliohjelmassa, jonka jälkeen asetti niitä käyttävän piirtofunktion päivitysfunktioksi. Ohjelman monimutkaistuessa tätä oli tarkoitus refaktoroida useampaan erilliseen solmuun/moduuliin, mutta sen kaiken oleminen vielä yhdessä aliohjelmassa helpotti varjostuksen testausta.

6.5 Mallin lataus, piirto ja ambientti valaistus

Käytössä oli valmis testimalli FBX-muodossa, joka sisälsi väritekstuurin, normaalikartan, verteksien paikka-, tekstuurikoordinaatti ja normaalidatan sekä indeksidatan. Käyttäen apuna skriptejä mallin verteksi ja indeksidata muutettiin binääriksi suoraan sellai-

seen muotoon, jossa sen pystyi lataamaan renderöijään ja sitä kautta näytönohjaimelle ja testitarkoituksia varten tekstuurit muunnettiin helposti ladattavaan Targa- eli .tga-formaattiin. Verteksi- ja indeksidataa käyttäen pystyttiin latausaikana luomaan normaalikarttaa varten tarvittavat tangentit ja bitangentit.

Mallin toimivuutta testattiin niin sanotulla pass-through-varjostimella, eli varjostimella, jotka ottavat syötetyn datan vastaan ja laittavat sen eteenpäin piirtojärjestelmässä tekemättä siihen muutoksia. Tätä oli kuitenkin helppo muuttaa laittamalla fragmenttivarjostimeen normalisoidun vakioarvon, jonka kanssa fragmentin väriarvo kerrottiin. Tämän avulla saatiin aikaan niin sanottu ambientti valaistus. Tämän yhteydessä pystyttiin laittamaan myös saamaan tekselien värien muuntaminen sRGB- ja RGB-väriavaruuksien välillä.

6.6 Valaistuksen ja efektien toteutus

Team Fortress 2:n 3D-malleissa voimakasta eroa pinnan värin tummuudella suhteessa pinnan orientaatioon suhteessa valonlähteeseen ei ole havaittavissa. Tämä tarkoittaa, että diffuusiovalaistus on hyvin vähäistä, joten vaikka se toteutettiin, sen vaikutus ei ollut merkittävä lopputuloksen kannalta.

Referenssi taidetyylin 3D-mallit ovat pienempiä yksityiskohtia kuten kumisaappaita lukuun ottamatta mattapintaisia, mikä viittaa siihen, että spekulaarivalaistusta on vähän, mutta vaikkei se näkynyt selkeästi, varjostin teki silti operaation. Valaistus tehtiin normaalimappia apuna käyttäen, joten ennen valaistusoperaatioiden suorittamista katsojan ja valonlähteen sijainti muutettiin tangenttiavaruuteen ja käytettiin verteksien normaalien sijaan normaalimapin normaaleita lopullisen valaistuksen laskemiseen.

Lopuksi mallin ääriviivoja vaalennettiin tekemällä operaatio, jossa mitä pienemmässä kulmassa pinta on suhteessa kameraan, sitä vaaleampi se on. Tämä sai aikaan joillekin sarjakuvamaisemmin varjostetuille peleille tyypillisen halo-efektin mallin ympärille. Efektejä havainnollistaa kuva 4.

Kuva 4. Valaistusten toteuttamisen eri vaiheet. Ambientti valo ilman teksturointia (uloimpana vasemmalla), teksturoitu malli diffuusiovalolla (sisempi vasemmalla), sama efekti kevyen spekulaarivalaistuksen kanssa (sisempi oikealla) ja lopullinen efekti (uloin oikealla).

6.7 Puurakenteen ja dataorientoituneisuuden toteuttaminen

Kun itse varjostuksen toteutus oli valmis, oli aika optimoida tuotosta. Tarkoitus oli jakaa moduuli, joka sisälsi kaiken renderöintilogiikan, useampaan moduuliin. Kuitenkin noudattaen dataorientoitunutta lähestymistapaa tehtiin arvio, jonka tuloksena oli se, että logiikka on liian yksinkertainen, että se saisi performanssietua moduuleihin jakamisesta, ja sen sijaan optimointi tehtiin datan asettelun suhteen katsoen, mikä on kohdealusta ja mitä operaatioita suoritetaan eniten.

Testitapauksen suppeudesta huolimatta rajapinnan valmius laajemmalle puurakenteelle saatiin toteutettua. Se latasi käyttäjän määrittelemät moduulit, ja niitten tarvitsemat resurssit. Käyttäjä pystyi myös itse kirjoittamaan moduuleja rajapinnalle tarvittaessa.

7 Loppusanat

Lopputuloksena oli toimiva ja helposti laajennettava renderöijä ja lupaava alku pelimoottorin kehitykselle. Rajapintaa tultaneen käyttämään jatkossa kokonaisten pelien tekemiseen, jonka yhteydessä sitä tullaan kehittämään eteenpäin ominaisuuksiltaan peliprojektien määrittämään suuntaan.

Optimoinnin puolesta renderöijän tilanne jäi vähemmän optimaaliseksi. Kuten jo projektin aloitusvaiheessa ennustettiin, rajapinta ja toteutettu esimerkkitapaus jäi yksisäikeiseksi, ja sen monisäikeistäminen tapahtunee vasta paljon myöhemmässä vaiheessa. Tällaisen monisäikeistämisessä haasteet ovat erityisesti vain pääsäikeeltä kutsuttavan

OpenGL-rajapinnan kutsujen hallinnointi ja pääsäikeen synkronointi muitten säikeitten kanssa.

Muistin allokoinnin ja puurakenteisuuden suhteen jäi parantamisen varaa tulevaisuudessa. Rajapinta toteutettiin niin, että jokainen moduuli allokoi itse tarvitsemansa määrän dynaamista muistia suoraan malloc-komennolla. Optimaalisimmassa tapauksessa joko ohjelma kutsuisi joko malloc-komentoa yhden kerran prosessin elinaikana tai ei ollenkaan. Tämän jälkeen ohjelma jakaisi moduuleille tarvitsemansa määrän muistia sisäisen muistin kautta. Tällainen muistin jakaminen ohjelman sisäisesti eri moduuleille, jotka käsittelevät itsenäisesti omaa muistiaan, on tyypillistä isoissa kaupallisissa pelimoottoreissa [2]. Lisäksi muistinvarausta voidaan optimoida muilla tavoin.

Joel Spolsky kirjassaan ”Joel on Software” suosittelee myös, että malloc-komennolla varattaisiin ainoastaan jonkin kahden potenssin verran tavuja. Tämä johtuu siitä, että muistin allokoijaa voidaan usein kuvata linkitettynä listana järjestelmämuistin lohkoista, eli niin sanottuna vapautusketjuna, joka malloc-kutsua kutsuttaessa käy läpi linkitettyä listaa etsien tarpeeksi suurta lohkoa. Sen löytäessään se jakaa lohkon kahteen osaan joista toinen on tasan pyydetyn muistimäärän pituinen ja toinen jäljelle jäävän osan pituinen. Tämä tarkoittaa sitä, että mitä enemmän ei-kahden potenssin pituisia lohkoja muistista varataan, sitä suuremmalla todennäköisyydellä listaan muodostuu väliin lohkoja, jotka ovat liian pieniä varattavaksi, joka puolestaan hidastaa seuraavia allokointikutsuja. [9, s. 11.]

Renderöijän puurakenteen toteutus, vaikka olikin toimiva käyttötarkoitukseen, jäi yksinkertaiseksi. Sitä olisi voinut parannella niin, että sen luonti olisi dynaamisempaa. Geneerisemmissä suuremman skaalan renderöijissä puun toteutus suositellaan monesti toteutettavan niin, että se luodaan jokaista kuvaa varten uudestaan [4]. Kuitenkin kun projektin skaala otettiin huomioon, päätettiin toteutus jättää myöhempään vaiheeseen.

Projektin skaalan suppeus aiheutti jonkin verran myös kovakoodattujen ominaisuuksien toteutusta. Tämä tapahtui siksi, että liiallista tulevaisuusvarmistelemista ja moduuleihin jakamista haluttiin välttää.

Jäi epäselväksi, ylittääkö vanhentuvan OpenGL-rajapinnan taaksepäin yhteensopivuudesta saatavat hyödyt Vulkanin eteenpäin yhteensopivuuden ja uusien ominaisuuksien edut. Vulkan-rajapintaa ollaan monin tavoin geneeristetty mahdollistamaan palvelimen

käytön laajemmassa skaalassa ja joustavammin, ja sisältää siksi vähemmän reaaliaikagrafiikkaspesifisiä ominaisuuksia. Tämä mahdollistaa palvelimen käytön myös renderöijän kannalta laajemmassa mittakaavassa; joitakin renderöijän operaatioita, joita ei aikaisemmin pystytty helposti suorittamaan palvelimella onnistuu nykyään ongelmitta. Kuitenkin pienemmässä mittakaavassa – kuten pienemmän mittakaavan renderöijän toteutuksessa – saattaa lyhyemmällä aikavälillä OpenGL:n käyttö antaa joitakin etuja nopeuttamaan tuotantoa ottaen huomioon, että rajapinta on vielä nykyisin monelle grafiikkaohjelmoijalle tutumpi ja että monia Vulkanin ominaisuuksista ei välttämättä edes tarvita pienemmän skaalan projekteissa. Kuitenkin koska kyseessä on nopeasti eteenpäin menevä ala, tulisi projektin jälkeen kartoittaa mahdollisuutta Vulkanin lisäämisestä yhdeksi renderöijän tukemaksi rajapinnaksi.

Kokonaisuudessaan yksinkertainen renderöijä täytti suurimmaksi osaksi projektin tavoitteet, ja sitä edeltäviä rajapintaa käyttäviä pelinkehitysprojekteja varten se osoittautui hedelmälliseksi kehitysalueeksi.

Lähteet

1 Wright, Richard S.; Haemel, Nicholas; Sellers, Graham & Lipchak, Benjamin. 2011. OpenGL SuperBible. Fifth Edition. Addison-Wesley.

2 Acton, Mike. 2014. Data-Oriented Design and C++. Luento. CppCon 2014.

3 Khronos Group. 2016. Vulkan 1.0.16 – A specification. Khronos Group.

4 Nicolas Guillermot. 2017. Design Patterns for Low-Level Real-Time Rendering. Luento. CppCon 2017.

5 2017. Render graphs and Vulkan. Verkkodokumentti <http://themaister.net/blog/2017/08/15/render-graphs-and-vulkan-a-deep-dive/> Luettu 07.11.2017.

6 Granberg, Carl. 2009. Character Animation with Direct3D. Charles River Media.

7 Valgrind Developers. Valgrind. Verkkodokumentti <www.valgrind.org> Luettu 18.04.2018

8 NVIDIA Corporation. Nvidia Graphics Debugger Documentation. Verkkodokumentti <https://docs.nvidia.com/linux-graphics-debugger/index.html> Luettu 18.04.2018

9 Spolsky, Joel. 2004. Joel on Software: And on a Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work With Them in Some Capacity. Appress.

Varjostuksen toteutuksen eri vaiheet

Suurennettu kuva efektien toteutuksen eri vaiheista. Ylhäällä vasemmalla on malli ambientilla valaistuksella ilman tekstuuria. Ylhäällä oikealla ja alhaalla vasemmalla oleva malli on teksturoitu ja ovat valaistu diffuusiovalaistuksella, ja vasemmalla alhaalla olevassa on kevyt spekulaarivalaistus. Alhaalla oikealla oleva malli on viimeistelty valaistus. 3D-mallin oikeudet omistaa Antti Sartanen.